“Performance bugs” and How to Find Them

Rune Erlend Jensen
MemoScale / Norwegian University of Science and Technology

Per Simonsen
MemoScale
Abstract

- “Performance bugs” are performance anomalies.
- We identify memory address aliasing as an important mechanism causing performance bias on modern Intel microarchitectures.
- We link this effect to memory allocators and demonstrate that they can hurt performance in several cases.
Presentation Overview

- Who are we?
- Optimization and data storage
- Optimization results
- Finding a "Performance Bug"
- Exploring the impacts
Who are the presenters?

- Rune Erlend Jensen, Performance engineer
  - Extensive optimization experience
  - Ph.D. candidate at the Norwegian University of Science and Technology.
- Per Simonsen, CEO, MemoScale
  - MSc Entrepreneurship/Computer Science
  - MSc Business Administration
MemoScale

- Start-up with strong ties to the Norwegian University of Science and Technology.
- *We leverage our expertise in optimization, software engineering and coding theory to solve bottlenecks and improve performance in data storage systems.*
Improving Performance

- Fastest unbroken SHA3 Candidate: Blue Midnight Wish
- Fastest Reed Solomon Erasure Coding implementation
- New erasure codes optimized for fast compute and rebuild
Chasing Bottlenecks

- From hardware to software
- Hardware getting faster
- The CPU is still the center player
Compute Intensive Components

- Erasure coding
- Hashing
- Encryption
- Compression
Test Setup – Benchmark of MemoScale vs Jerasure and Intel ISA-L

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data blocks</td>
<td>10</td>
</tr>
<tr>
<td>Redundancy blocks</td>
<td>4</td>
</tr>
<tr>
<td>Block size</td>
<td>4 MB</td>
</tr>
<tr>
<td>Number of cores/threads</td>
<td>1</td>
</tr>
<tr>
<td>Measurement criteria</td>
<td>Data encoded excl. redundancies, MB/s</td>
</tr>
<tr>
<td>Codes tested</td>
<td>Reed Solomon and MemoScale compute optimized EC</td>
</tr>
<tr>
<td>Libraries tested</td>
<td>Jerasure, Intel ISA-L, MemoScale</td>
</tr>
</tbody>
</table>
EC Encoding Speed I – Intel Processor

**Processor:**
Intel Xeon
E5-2676 v3
2.4 Ghz

**Erasure coding (EC):**
* Reed Solomon
** MemoScale compute optimized

![Graph showing MB/S Data encoded for different encoding methods](image)

- Jerasure*
- Intel ISA-L*
- MemoScale*
- MemoScale CO**

**Processor:**
Intel Xeon
E5-2676 v3
2.4 Ghz

**Erasure coding (EC):**
* Reed Solomon
** MemoScale compute optimized
EC Encoding Speed II – ARM Processor

**Processor:**
HiSilicon Kirin 620 processor, ARM Cortex-A53 64-bit SoC 1.2ghz

**Erasure coding (EC):**
* Reed Solomon
** MemoScale compute optimized

![Graph showing EC Encoding Speed II – ARM Processor](image_url)
EC Encoding Speed III – AMD Processor

**Processor:**
AMD RYZEN 7 1700 8-Core 3.0 GHz (3.7 GHz Turbo)

**Erasure coding (EC):**
* Reed Solomon
** MemoScale compute optimized

* Jerasure
** MemoScale
** MemoScale CO

MB/S Data encoded

- 0
- 2000
- 4000
- 6000
- 8000
- 10000
- 12000
- 14000
- 16000

0 2000 4000 6000 8000 10000 12000 14000 16000

Jerasure* MemoScale* MemoScale CO**
Finding "Performance Bugs"

- What is a "Performance Bug"?
- Optimizing for performance
Measurement Bias I

- What if changing your user name affected the execution time of programs?
Measurement Bias II

- What if changing your user name affected the execution time of programs?
  - Runtime performance change from external factors, introducing bias.
    - Environment variables
    - Link ordering
Measurement Bias III

- What if changing your user name affected the execution time of programs?
  - Changes in memory layout interacting with various hardware mechanisms.
    - Hard to predict :(
Measurement Bias IV

- Challenge for performance analysis
  - Comparison of algorithms on different system configurations can give misleading results.
  - Effects can invalidate perceived speedup (Mytkowicz et al. [1]).
Methodology I

- Control execution
  - Control execution environment (disable ASLR*).
  - Instrument program using hardware performance counters.
  - Measure over range of environments.

*Address space layout randomization.
Methodology II

- Memory execution context, assuming a 64-bit process running on a Linux system.

Program code and static data

```
0x7fff'ffffffff
env. vars
stack
mmap area
heap
bss
data
text
0x400000
```

2017 Storage Developer Conference. © MemoScale. All Rights Reserved.
Bias from Environment Size I

- A simple program*, with no surprises?

```c
static int i, j, k;

int main() {
    int g = 0, inc = 1;
    for (; g < 65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}
```

*Microkernel first presented by Mytkowicz et al.[2], showing env. bias.
Bias from Environment Size II

- Execute program under different environments.

![Graph showing the number of cycles against the number of bytes added to the environment.]
Bias from Environment Size III

- How to find the cause?
  - Accurate measurements
    - Hardware performance counters
  - Use correlation
    - Check *lots* of different CPU performance events
    - Correlate them with performance
## Bias from Environment Size IV

<table>
<thead>
<tr>
<th>Performance counter</th>
<th>Median</th>
<th>Spike 1</th>
<th>Spike 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld blocks partial.address alias</td>
<td>137</td>
<td>327,871</td>
<td>327,848</td>
</tr>
<tr>
<td>cpu clk unhalted.thread p</td>
<td>692,686</td>
<td>825,311</td>
<td>823,872</td>
</tr>
<tr>
<td>uops executed port.port 0</td>
<td>240,248</td>
<td>106,477</td>
<td>106,409</td>
</tr>
<tr>
<td>uops executed port.port 1</td>
<td>249,016</td>
<td>100,913</td>
<td>101,250</td>
</tr>
<tr>
<td>uops executed port.port 2</td>
<td>336,768</td>
<td>345,932</td>
<td>346,328</td>
</tr>
<tr>
<td>uops executed port.port 3</td>
<td>358,848</td>
<td>356,919</td>
<td>357,723</td>
</tr>
<tr>
<td>uops executed port.port 4</td>
<td>280,766</td>
<td>276,036</td>
<td>276,199</td>
</tr>
<tr>
<td>uops executed port.port 5</td>
<td>251,631</td>
<td>121,517</td>
<td>121,571</td>
</tr>
<tr>
<td>uops executed port.port 6</td>
<td>234,337</td>
<td>178,586</td>
<td>178,646</td>
</tr>
<tr>
<td>uops executed port.port 7</td>
<td>137,428</td>
<td>130,325</td>
<td>130,542</td>
</tr>
<tr>
<td>resource stalls.any</td>
<td>274,373</td>
<td>406,364</td>
<td>406,256</td>
</tr>
<tr>
<td>resource stalls.rs</td>
<td>272,371</td>
<td>135,567</td>
<td>136,503</td>
</tr>
<tr>
<td>cycle activity.cycles ldm pending</td>
<td>590,879</td>
<td>718,973</td>
<td>721,580</td>
</tr>
<tr>
<td>cycle activity.stalls l2 pending</td>
<td>465,205</td>
<td>490,004</td>
<td>491,656</td>
</tr>
</tbody>
</table>
## Bias from Environment Size IV

<table>
<thead>
<tr>
<th>Performance counter</th>
<th>Median</th>
<th>Spike 1</th>
<th>Spike 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id blocks partial.address alias</td>
<td>137</td>
<td>327,871</td>
<td>327,848</td>
</tr>
<tr>
<td>cpu clk unhalted.thread p</td>
<td>692,686</td>
<td>825,311</td>
<td>823,872</td>
</tr>
<tr>
<td>uops executed port.port 0</td>
<td>240,248</td>
<td>106,477</td>
<td>106,409</td>
</tr>
<tr>
<td>uops executed port.port 1</td>
<td>249,016</td>
<td>100,913</td>
<td>101,250</td>
</tr>
<tr>
<td>uops executed port.port 2</td>
<td>336,768</td>
<td>345,932</td>
<td>346,328</td>
</tr>
<tr>
<td>uops executed port.port 3</td>
<td>358,848</td>
<td>356,919</td>
<td>357,723</td>
</tr>
<tr>
<td>uops executed port.port 4</td>
<td>280,766</td>
<td>276,036</td>
<td>276,199</td>
</tr>
<tr>
<td>uops executed port.port 5</td>
<td>251,631</td>
<td>121,517</td>
<td>121,571</td>
</tr>
<tr>
<td>uops executed port.port 6</td>
<td>234,337</td>
<td>178,586</td>
<td>178,646</td>
</tr>
<tr>
<td>uops executed port.port 7</td>
<td>137,428</td>
<td>130,325</td>
<td>130,542</td>
</tr>
<tr>
<td>resource stalls.any</td>
<td>274,373</td>
<td>406,364</td>
<td>406,256</td>
</tr>
<tr>
<td>resource stalls.rs</td>
<td>272,371</td>
<td>135,567</td>
<td>136,503</td>
</tr>
<tr>
<td>cycle activity.cycles ldm pending</td>
<td>590,879</td>
<td>718,973</td>
<td>721,580</td>
</tr>
<tr>
<td>cycle activity.stalls l2 pending</td>
<td>465,205</td>
<td>490,004</td>
<td>491,656</td>
</tr>
</tbody>
</table>
The cause: 4K Address Aliasing I

- Addresses of static \( i, j \) and \( k \) are fixed for all environments.
The cause: 4K Address Aliasing II

- Addresses of \textit{inc} and \textit{g} live on the stack.
- The stack is pushed down by the environment variables!
The cause: 4K AddressAliasing III

- Spike when address of `inc` is aliasing with `i` in the last 12 bits.
  - The last 3 hex digits.

```
USER=lars;SHELL=/bin/bash;X=0000
00000000000000...
```

```
0x601044
0x601040
```

Collision
The cause: 4K Address Aliasing IV

- Performance impact from *false* dependencies in CPU.
- Hurts Out-of-Order execution.

<table>
<thead>
<tr>
<th>USER=lars;SHELL =/bin/bash;X=0000 00000000000000...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Collision

```
0x7fff'ffffe03c
0x7fff'ffffe038
```

```
0x601044
0x601040
0x60103c
```
How to Measure Address Aliasing

- Hardware counter event
  - Linux:perf
    - Manually add event type

LD BLOCKS PARTIAL.ADDRESS_ALIAS
“Counts the number of loads that have partial address match with preceding stores, causing the load to be reissued.” [3, B.3.4.4]
Other architectures I

Core

Nehalem (HT)
Other architectures II

Ivy Bridge
Bias from Heap Allocation I

- Alias conflicts can happen between any sections of memory.
- While stack addresses depend on environment, heap addresses depend on allocators.
Bias from Heap Allocation II

- Addresses returned by different heap allocators when allocating pairs of equally sized buffers.
- Same 12 bit suffix indicate an aliasing pair.

<table>
<thead>
<tr>
<th></th>
<th>5,120 B</th>
<th>1,048,576 B</th>
</tr>
</thead>
<tbody>
<tr>
<td>glibc (ptmalloc)</td>
<td>0x602010</td>
<td>0x2aaaaaaaf6010</td>
</tr>
<tr>
<td></td>
<td>0x603420</td>
<td>0x2aaab096010</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>0xe0e000</td>
<td>0xe0e000</td>
</tr>
<tr>
<td></td>
<td>0xe10800</td>
<td>0xf0e000</td>
</tr>
<tr>
<td>jemalloc</td>
<td>0x2aaaabc0e000</td>
<td>0x2aaaabc06000</td>
</tr>
<tr>
<td></td>
<td>0x2aaaabc10000</td>
<td>0x2aaaabd06000</td>
</tr>
<tr>
<td>Hoard</td>
<td>0x2aaaaab00070</td>
<td>0x2aaaaab00070</td>
</tr>
<tr>
<td></td>
<td>0x2aaaaab02070</td>
<td>0x2aaaaabf40070</td>
</tr>
</tbody>
</table>
Bias from Heap Allocation III

- Most allocators tend to use anonymous memory mapping for large requests.
  - This performed by mmap.
  - Always at a page boundary.
- Since page size is 4 KiB, such allocations will always alias with each other.
Bias from Heap Allocation IV

- Other reasons for 4K alignment
  - RDMA
    - Requires page locked memory
      - Naturally aligned to 4K
  - Drives use 4K blocks (not the same, but…)
  - Memory mapped files
    - mmap – always page aligned
Optimization Resistant Code I

- First example program was too simple?
- Simplified implementation of convolution with a fixed kernel?
Optimization Resistant Code II

static float k[5] = {0.1, 0.25, 0.3, 0.25, 0.1};

void conv(int n, const float *in, float *out) {
    int i, j;
    for (i = 2; i < n - 2; ++i) {
        out[i] = 0;
        for (j = 0; j < 5; ++j) {
            out[i] += in[i-2+j] * k[j];
        }
    }
}
The key property is the data access pattern:

- In buffer
- Sliding window
- Out buffer
Optimization Resistant Code IV

- Adjust address of output array manually.
  - Offset arrays by $d$ elements.
    - $\text{conv}(N, \text{in}, \text{out} + d)$;
    - $N = 2^{20}$ (4 MiB data)
- Evaluate performance.
Optimization Resistant Code V

Haswell -O2

- Cycles
- Alias

Haswell -O3

- Cycles
- Alias
Aliasing in BLAS* libraries I

- What about heavily optimized libraries?
- Measured matrix-vector multiplication (cblas dgemv) with increasing offset between buffer addresses. Computing $\mathbf{y} = \mathbf{A}\mathbf{x}$.

*Basic Linear Algebra Subprograms
Optimizing for address aliasing

- Mark buffers with `restrict`
  - Can allow the optimizer to reduce number of memory accesses.
- Use optimized libraries
  - They might mitigate the effect.
Optimizing for address aliasing II

- Use a special purpose allocator.
  - Heuristics for avoiding pairwise aliasing buffers.
  - No known implementation with this goal in mind, to our knowledge 😞
Optimizing for address aliasing III

- Explicitly adjust address offset.
  - Consider memory address alignment for performance tuning inner loops.
  - Buffer addresses as candidate for automatic performance tuning.
Learning points I

- Accurate benchmarking is hard.
  - Many uncontrolled effects creates bias.
  - CPU’s are very complex.
- Optimizing is hard.
  - Too many rules to address at the same time.
  - Performance is a sum of good and bad cases.
Learning points II

- We focused on address aliasing.
  - And some of the implications it had.
- There are more (and bigger) issues like this.
  - CPU cache!
  - Multithread locks 😞
  - TLB (Translation lookaside buffer).
Learning points III / Summary

- Accurate benchmarks are crucial.
  - Large set of benchmarks.
  - Many measurement metrics needed.
    - Not only timings.
  - Identify performance issues (correlation).
    - Address aliasing (in our case).
References


Further Questions…?

rune.jensen@memoscale.com
per.simonsen@memoscale.com